A System for Building Scalable Parallel Applications

Simon Dobson[*]
Andy Wellings

Department of Computer Science, University of York, Heslington, York YO1 5DD, UK.

Abstract

One of the major problems with programming scalable multicomputer systems is defining appropriate abstractions for the programmer. In order to allow applications to scale their resource consumption according to run-time conditions, we propose a view of a scalable system which treats memory as a collection of programmer-defined and -extensible data structures. These structures may be transparently distributed across the nodes of the system while still presenting a single-entity abstraction to the programmer. The implementation of such structures, as a kernel for a programming environment, are presented, along with examples of their use.

1. Introduction

Experience has shown that the effective use of parallel systems often forces programmers to deal with low-level architecture-specific features of particular hardware platforms and communications protocols: this both reduces the portability of software and increases the burden on the programmer. This situation is exacerbated by *scalable* parallel systems, where the machine's underlying hardware resources may change dynamically. What is needed is an environment in which a high-level view of the scalable computer – having global memory and unlimited parallelism – is mapped transparently onto the underlying multicomputer architecture.

This paper describes a programming environment designed specifically for exploiting the potential of highly parallel, highly scalable computers. Section two describes some of the problems encountered when attempting to design a programming environment for a scalable computer, and suggests that the major problem is essentially one of hiding the underlying disjoint memory architecture. In order to solve this problem, section three proposes the use of "scalable memory" built from high-level data structures which can transparently span processing nodes. Section four discusses the use of such memories to automatically regulate concurrency. Section five contains some examples of our system in use, whilst section six presents our conclusions.

2. Scalable Computing

We will first define what is meant by scalability, in terms of architectures, operating systems, and applications, and then discuss some of the problems which it introduces.

[*] Supported by a research studentship from the UK Science and Engineering Research Council.

2.1. What is Scalability?

A *scalable* computer is a machine to which additional memory and processors may be added so that the machine's power may be increased (or decreased) as desired. Although shared-memory multiprocessors may be described as scalable, contention for the shared memory places a limit on the number of processors which may be used. Workstations connected by a broadcast network are similarly limited by the saturation point of the shared communications medium. Multicomputers – collections of processor-memory pairs connected by a sparse point-to-point network – offer a better solution, as they have no globally-shared resources to place an upper-bound on their size.

Operating system and programming environment design are often seen as exercises in abstraction. For a scalable system, this suggests that the physical resources of the machine are hidden from users and programmers – doing otherwise means that programs must be re-written (or at least re-compiled) if the system's resources are changed. A scalable operating system abstracts away from notions of process location, and allows applications composed of a number of processes to take advantage of the system's inherent parallelism.

2.2. Problems of Scalability

For the applications programmer, writing a program which can take advantage of whatever resources are available at run-time is a great challenge. It effectively implies a very flexible attitude towards *entity*, *memory* and *process* management.

Entity Management

Entity management – how elements of a system's software are named and accessed – must take account of the distributed nature of the underlying hardware. It is desirable to abstract away from the physical locations of such *software resources*, allowing different resources to interact regardless of their relative locations. This permits remote resources to be manipulated in an identical fashion to local ones, and relieves the programmer of the need to manage data locality explicitly. Techniques for achieving such *location transparency* vary from the channel abstraction of Occam[19] to the shared tuple space of Linda[1].

An approach which is gaining popularity is the *virtual object space*[18], which uses object-oriented techniques to name and manipulate software resources. Object names allow objects to be located and accessed regardless of their location in the network. Objects are then the unit of distribution, with each object residing on exactly one node at any time (although objects may migrate between nodes in some systems, notably [13]).

Memory Management

In discussing memory management, one must consider two distinct ideas: the way in which local memory is managed, and the way in which these local memories are used collectively.

Local memory is bounded, and this introduces problems of managing single entities which are larger than a single local memory. The use of techniques such as distributed shared virtual memory[15] – in which physical memories are used as caches for virtual pages stored on disc – introduces consistency and scalability problems, and for this reason many systems keep to a *real memory* architecture. Nonetheless, it is obviously desirable to allow applications to use the system's memory as a single unit, without worrying about its topology and distribution, as this allows these architectural features to be changed without impacting on correctly-written applications.

Process Management

Since an application has no information at compile-time about the processing requirements which will be available at run-time, concurrency regulation – deciding how many processes to create for a particular task, and where to locate them – is difficult. In the multiple-worker style of concurrency, for example, it would be advantageous to allow the system to determine the number of workers to create: this number could then be tailored to run-time conditions, such as how many processors the application is using.

2.3. Phoenix: an Environment for Scalable Parallel Programming

We have developed a scalable parallel operating system nucleus called *Wisdom*[3,20,21,22], and are now developing a programming environment called *Phoenix*[1] for use within it. The aim of Phoenix is to encourage and simplify the task of writing highly parallel, highly scalable, general-purpose applications. It takes the form of an object-oriented class library and associated supporting tools.

In order to support scalability, Phoenix views “memory” as being a set of high-level *typed memory modules* which the programmer may instantiate as required, independent of the underlying network architecture[6]. These modules appear to the programmer as familiar data structures: whilst objects remain the units of *distribution*, the memory modules are the units of *aggregation*. Any memory module may be as large as desired, unconstrained by local memory sizes: a large structure will distribute itself as necessary according to run-time conditions whilst still behaving as a single object from clients' viewpoints. Concurrency regulation occurs through these modules, with multiple worker processes being attached to memories. The processes are then replicated according to the distribution of the memory. It is thus possible to create and manipulate structures of an arbitrary size, and to process them in parallel, without being concerned with the exact mechanism or policy of their distribution – although the policy may be tailored if required to suit particular applications.

The remainder of this paper describes the mechanisms used to realise these ideas.

3. Scalable Memory

The implementation of scalable memory essentially involves the hiding of exactly *what* data is located in exactly *which* physical address space. Furthermore, it is essential that the contents of memory are structured in a meaningful (to the programmer) manner, and that the hiding of location does not destroy the performance of the application.

3.1. Data Structuring

It is first worth considering exactly what form of data structuring is used within typical distributed applications. This then allows us to provide, at the base level of the programming environment, the most useful structures as fully distributed, scalable modules, whilst making it easy for programmers to expand the basic forms into specialised application-level structures. Since we are considering general-purpose computing, we cannot restrict our considerations to a particular programming domain: we must instead consider the whole range of applications which are (or might benefit from being) run on parallel machines.

We would contend that, whilst there are as many types of data structuring as there are programs, three forms of large-scale data structure predominate: arrays, associative memories and graphs.

[1]In mythology, the Phoenix is the keeper of Wisdom.

3.2. Requirements

All data structures in our system share a common set of requirements, although the exact manner in which these requirements are achieved differs between different categories of structure. The overall aims may be given as follows:

- a *single object* abstraction, so a structure may be treated as a single logical unit;
- *transparent distribution*, so the distributed nature of a structure, and the locations of its elements, do not interfere with algorithms (although they may be controlled if necessary);
- *unbounded size*, so that a structure may be as large as the application requires, unconstrained by the size of individual node memories.
- *variable size*, so a structure's use of storage is governed by the number of elements which it contains: small structures to not use storage unnecessarily;
- *lack of bottlenecks*, so parallel access is possible without too much interference between activities;
- *strong typing* to prevent abuse of structures; and
- *easy extension* to encourage re-use of existing code and to simplify the construction of application-specific structures.

In order to implement such structures, we have developed a model of large data aggregates called the *partitioned object model*[8], together with a number of implementation techniques specific to each category of memory discussed above.

3.3. A Partitioned Model of Objects

The partitioned object model is founded on the observation that, whilst data aggregations are best considered as objects like any other, they differ from “simple” objects (such as integers, points *et cetera*) in possessing a highly organised and regular internal structure. Our model utilises this feature to control the distribution (or *partitioning*) of data between various host nodes in a manner which is transparent to clients of the structure. In this way, a single logical software resource may be composed from a number of objects[7] in a manner which is largely invisible to applications programmers.

A Basic Distributed Object Model

The basic object model used by Phoenix is very simple. The system itself is written in a dialect of C++[24] which has been extended to deal with distribution and concurrency: we therefore take the C++ model of objects as our base.

Object names take the form of communications capabilities, so that a capability uniquely identifies an object anywhere in the network. This allows two objects to interact regardless of their relative locations. The fact that objects may reside on different nodes introduces some additional restrictions: specifically, instance variables should not be accessible from outside the object. An object resides on exactly one node[2].

[2]We do not outlaw object migration, although our current implementation does not provide this feature.

Object interactions take the form of calls to methods. Method invocation is performed exactly as expected, using remote procedure call[5] techniques to translate C++-style calls into request-and-reply message transfers. It follows the same semantics as in C++, but is extended with the notion of an *asynchronous* method which never returns and does not block the caller. Asynchronous methods are represented by the pseudo-type `async`: the asynchronous nature of a method is defined by the object's class, and it is not possible to call an arbitrary method asynchronously (thus removing the need for "future" returned values[12,16]).

Objects are multi-threaded, so several method calls may be active at any time within a single object. Support is provided for intra-object concurrency control using "lock" objects, which are a fusion of ideas from the Arjuna[23] and DRAGOON[2] projects allowing concurrency control *via* deontic logic.

The Phoenix class library is formed from a single tree, with the root being the class `Object`. All `Object`s thus share a common core protocol: amongst the functions provided are equality testing and copy generation, which call virtual functions defined by the programmer for each sub-class. Little use is made of multiple inheritance.

An Overview of Partitioning

Partitioned structures are "collections" in the Smalltalk sense[11], representing what would normally be termed data structures. Each structure is a software resource whose purpose is to allow access to other objects (its *elements*) according to some meaningful protocol.

A *partitioned collection* is a collection whose size is not constrained by the size of a single node memory. It is composed of a number of smaller *component* objects, each holding a small number of the full collection's elements. These components are distributed between nodes, and are linked by a tree of *partition* objects which manage distribution. The components form the leaves of the tree, the partitions the branches, so every component is attached to exactly one partition object.

The tree may be fixed, or may vary with time. For an array, whose size is fixed at its creation, the tree will be created and will never change; for associative memories and graphs, the number and arrangement of both partition and component objects will change as elements are added and removed.

A partitioned collection has the property that any component acts as a *pseudonym* for the entire collection. A request for any element of the structure may be addressed to any component, with the same result: alternatively, all components can deal with exactly the same set of requests. Accesses to local and remote data are identical, with the only difference being that the time taken to satisfy requests for remote data will be longer than for local data. Thus the collection behaves as a single resource, even though it is composed of a number of independent objects. A generic partition tree is shown in figure 1.

Requests for data are satisfied by a process termed *resolution*. When a request is sent to a component (by another object or by a sub-class), it is checked to see whether it can be satisfied locally. If it can, then servicing occurs locally; if not, the details of the requested element are passed to the partition. The partition then makes use of a distributed algorithm to locate the component which *can* service the request locally, and the original request is forwarded *in toto* to this component. There is no requirement for global locking, so several requests may be active within the structure at any one time – only if two requests access the the same object will concurrency control become necessary.

There is thus a very clear division of responsibility: components perform data processing, whilst partitions perform data location. A component's knowledge of distribution is limited to a single "locality" test, with the partition being free to use any policy for distribution it wishes; conversely, a partition only sees enough of requests to perform resolution, and does not perform any data processing. It is a simple matter to use a specialised partition sub-class to tailor distribution of data, or to sub-class a component class to provide advanced processing.

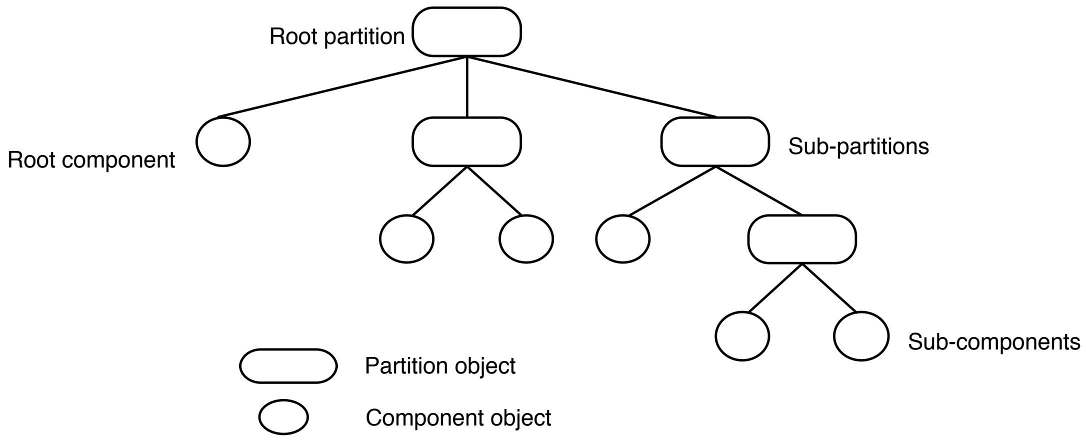

Figure 1: A generic partition tree

3.3.1. Some Partitioning Techniques

As already stated, we consider three categories of structures – arrays, associative memories, and graphs – to be fundamental. However, the internal structures of these categories is obviously very different. Within the framework of partitioned collections, we have therefore developed appropriate category-specific implementation techniques. Care has been taken to ensure that a clean interface is retained between sub-classes, and between collection and partition objects, to maximise the possibilities for code and design re-use.

Arrays

An array represents a bounded discrete *n*-space. It may thus be defined by a “region” specifying its boundaries. It may further be observed that such a region may be built from a number of smaller regions, which together define the same space. Partitioning an array is a matter of dividing the “global” region into a number of smaller sub-regions, distributing them, and creating a component array for each.

The partitioning process occurs when an array is first created, with the first component generating a partition object. The partition splits the global region into a number of smaller sub-regions, each of which will eventually be used by a component to define its local storage.

The partition determines which regions to hold locally, and how many sub-partitions to create. It then creates the sub-partitions and divides the remaining sub-regions between them. The sub-partitions will then in turn keep some regions and generate further sub-partitions, until the process bottoms-out with a partition retaining all sub-regions itself. A partition thus knows all the regions which are held below it in the tree. A component is generated for each locally-held region, with the size of the array being taken from its region.

A `Region` is an object representing a small region of an array's space. It is capable of “flattening” a point into an offset, allowing multi-dimensional regions to be stored in simple linear arrays. The protocol for the `Region` class is shown in figure 2. The important functions provided by `Region` are defined as virtual member functions, so that they may be over-ridden in sub-classes to provide specialised `Regions`.

```
class Region : public Object {
  ...
public:
 Region( Point ll, Point our );

 virtual int Size( void );
 virtual boolean Contains( Point p );
 virtual int Flatten( Point p );
 virtual Point Expand( int i );
 virtual int Split( int md, Region **&subRegions );
};
```

Figure 2: The Region protocol

Resolution makes use of the fact that every element which might be requested is a member of exactly one region, and a partition can determine which object – locally-held component or sub-partition – a region was assigned to, if any. Elements are addressed using “points” within a region. Hence a request to locate a particular element proceeds as follows: the partition's region table is searched to locate the region containing the requested point. If this region maps onto a locally-held component, resolution has succeeded; if it maps onto a sub-partition, the request is forwarded to this partition for further resolution; if a region cannot be found, it may be assumed to be serviceable higher-up the tree and may be forwarded to the parent partition for further resolution.

Associative Memories

An associative memory is based around the principle of hashing. However, in a scalable environment, the standard hashing techniques do not work as the number of “buckets” of data is fixed, as is a bucket's size. What is required is an “extensible” hashing algorithm, and a suitable algorithm has been derived from similar methods used in storage management applications (for instance [9,10,14,17]). In effect, an associative memory is composed of a variable population of buckets, whose number varies as the structure grows or shrinks whilst maintaining the validity of keys.

When initially created, an associative memory will consist of a single bucket into which all elements are placed. When the bucket become full, it is split. A prefix of the hash key (say two bits) is taken, a partition with the correct number of new buckets created (three in this case, yielding four in combination with the original), and the elements of the split bucket divided between the new buckets. All these buckets share a common prefix of the hash key, but have a different two-bit suffix.

This process of splitting a full bucket occurs recursively: every time a bucket become full, another prefix of the key is consumed to re-distribute the elements. Conversely, if a number of buckets are empty, they may join to form a single bucket. The depth of the partition tree (and hence the size of the collection) may thus vary dynamically according to the number of elements in the structure.

Resolution occurs by examining the hash key of the sought element and determining its prefix. If this prefix is held locally, the sought element should be local; if not, the key is passed to the partition. This may then determine, from comparison of the key prefix with its own, whether the bucket containing this key is above or below in the tree, and forward or resolve the request accordingly. This comparison of hash keys requires no global information about the collection and so is a completely distributed algorithm.

Graphs

A graph is composed of a number of nodes, connected by arcs. Retrieval of elements is based on traversal of arcs from another, already-known node. Partitioning, in this case, is simply a matter of ensuring that nodes are evenly distributed; resolution is automatic, and requires no intervention by the partition.

Nodes are connected by arcs, which may be labelled and may have an associated direction. Node creation occurs by taking an existing node and generating an arc to a new, non-existent node. The calling node's partition object will then determine the location for the new node and perform creation and arc linking.

3.3.2. Affecting Distribution

The manner in which elements of a distributed application are mapped onto the underlying processing nodes has a major impact on performance, but is in many sense a peripheral issue: a distributed object-oriented program will work no matter how this mapping occurs, but some configurations will be more efficient than others.

The partitoning process allows programers to ignore the manner in which components are mapped onto nodes, or allows them to implement “designer” strategies for particular applications. There is, however, a third alternative: by allowing programs to acquire some of their distribution parameters at run-time, a particular partitioning strategy may be “fine tuned” simply by supplying different parameters.

We have implemented this option by providing a *property sheet*. This is a set of name-value pairs which may be interpreted as directives about how to partition a collection (or any other run-time property, for that matter). The intention is to place in the property sheet information which may affect the *distribution* of a program without affecting its *semantics*. Properties may be specified on a system-wide, per-class or per-instance basis, allowing a very fine degree of control to be exercised over an application.

Typical properties used at present control the width and depth of the partition tree, the number of elements held in each component, the “eagerness” with which the partition tree grows *et cetera*. Changing these parameters can drastically alter the distribution pattern of a collection without altering its code or semantics.

4. Processing in a Scalable Environment

The problems of concurrency *control* are well-understood, but concurrency *regulation* – determining how many processes to create for a given task, and where – has received far less attention.

We take the view that we may use our scalable memory to regulate our scalable processing. The basis for this assumption is that, in the partitioned model, a larger structure is distributed more widely (amongst more components and on more nodes) than a smaller structure, and the amount of data to be processed is a good metric for determining the amount of concurrency to employ.

4.1. What is a Process?

Within our object model, a method may be called asynchronously and give rise to a parallel thread of activity. This however is a rather unstructured view of concurrency: it is easy to see that unrestrained use of such methods could result in a serious danger of deadlock, interference, race conditions *et alia*.

```
class Activity : public Object {
private:
  ...

protected:
  virtual void Body( void );

  Activity( Collection *c, Group *g );

public:
  void SetAttachment( Collection *c );
  Collection *GetAttachment( void );

  async Start( void );
};
```

Figure 3: The Activity class protocol

In order to impose a structure onto processing, we define an `Activity` class of object. `Activity` objects are *in no way* privileged over other objects: they simply possess a protocol and support for attachment to our scalable memory modules. The basic protocol for activities is shown in figure 3. The `Body()` member is over-ridden in each sub-class to define the process' actions.

Such processes are intended to function in a “multiple worker” style of concurrent processing. That is to say, a number of worker tasks are each assigned a disjoint part of a data set, which they transform in some way. This transformation may require access to the data allocated to other workers. Concurrency regulation, then, is the task of determining how many workers to create, and where. There must also be support to enable a worker to determine its part of the overall data set.

4.2. Attachment and Concurrency Regulation

At the most basic level, a “process” accesses a “memory” in order to acquire (and possibly change) data. Since we support the notion of multiple memories (partitioned collections), a process must be *attached* to the particular structure which it is to process.

Attachment consists of informing the memory that an activity is to be attached to it. In general, the memory will then replicate the given process, once per component, with each replica being passed the name of the component with which it is co-located. The effect of this strategy is that as many processes will be created as there are components of a structure. The names of these replicas are collected into a `Group` object, which is a sub-class of `Activity` and allows all the replicas to be manipulated *en masse*. Activities notify their `Group` of state changes such as termination, making it possible to perform synchronisation.

In terms of programming style, processing a partitioned collection consists of instantiating a single process of the appropriate class and attaching it to the desired collection. From an activity's viewpoint, it will have access to the entire partitioned collection, using the same syntax, *via* the component to which it is attached. Since it must only actually process the local part of the component, there must be a method by which it can obtain the local data. For this purpose, all partitioned collections support the notion of *iteration*: a process may acquire the “first” item in a component and repeatedly request the “next” until the component is fully processed. For generality it is also possible to iterate globally across the entire collection. Component classes may also define alternative ways to acces local data – for example by allowing clients of an array to access the local Region.

```
class ObjectArray : public Array {
private:
  ...

protected:
  /* internal storage management */
  virtual void BasicAtPut( Point p, void  *o );
  virtual void BasicAt( Point p, void *o );

  /* resolution */
  virtual Array *BasicResolve( Point p );

public:
  Array( Region *gr );

  void SetLocalRegion( Region *lr );
  Region *GetLocalRegion( void );

  /* bounds and locality testing */
  boolean IsHeldLocally( Point p );
  boolean CanResolve( Point p );

  /* user-level access routines */
  void AtPut( Point p, Object *o );
  void AtCopy( Point p, Object *o );
  Object *At( Point p );

  /* partitioning */
  Partition *GetPartition( void );
  virtual void Initialise( Object *o =nil );

  /* concurrency */
  Group *Attach( Activity act );
  void AttachStartAndAwait( Activity *act );
  void AttachAndStart( Activity *act );
};
```

Figure 4: The ObjectArray class protocol, slightly compressed

If a process accesses only the local data, it will make maximum use of the principle of locality of reference no remote access to data will occur, hence no network traffic and no interference with other processes. In practice it is often necessary for a process, whilst dealing with local data, to make reference to other areas of the structure. Because of the transparency of access to data in partitioned collections, these accesses have exactly the same syntax and semantics as local accesses: remoteness has a *performance* aspect, but no *semantic* effect.

5. Examples

We are currently finishing the implementation of the Phoenix prototype, and are starting its evaluation *via* a series of example applications. The purpose of the evaluation is four-fold: to show that the “data-based” style of programming is indeed suitable for scalable systems; to ensure that the partitioning techniques we have developed are indeed scalable in real cases; to determine exactly how much performance is sacrificed by using resolution; and to investigate the effects of different distribution policies on test applications. We are also interested in the uses of caching the results of recent resolutions in order to accelerate the resolution process (using the techniques developed by Austin[4]).

We will now present examples of Phoenix code. The code is slightly simplified over that found in the actual system: the class protocols have been stripped-down to essentials, and the syntax used is "pure" C++ rather than the slightly embellished code required by our (primitive) RPC manager. We have also "flattened" the inheritance tree to remove abstract classes.

The two examples show the basic use of partitioning from a client's viewpoint, and the considerations in creating specialised collections and distributions. Both take the form of a cellular automaton simulation[25], which are discrete simulations based around arrays, with each element in the array representing a "cell." The values of the cells are updated through time, with each cell being updated according to the values of itself and its near-neighbours. They thus exhibit substantial locality of reference, and so are well-suited to execution within our environment. Automata of the form shown here are used extensively in electric field simulations.

Basic Partitioning

The skeleton protocol for an array of objects, `ObjectArray`, is shown in figure 4. For the sake of brevity we have compressed the class hierarchy, placing some functions (which are inherited from abstract superclasses in Phoenix) into a single class definition – the use of abstract superclasses maximises the possibilities for re-use and re-definition, at the expense of a proliferation of class definitions.

Clients call the `Array()` constructor to create an instance of the array. The `Initialise()` member is then called to perform the partitioning operation: the result is that the initial array instance is augmented by a partition tree and additionalcomponents. The original instance will have the same status as any other component, and will hold a small number of elements. Access to the collection may then occur through this component.

Each element of the array, in our example, is an instance of class `Cell`. A cell's value changes according to the current simulation time:

```
class Cell : public Object {
private:
  ...

public:
  Cell( int history );

  float GetValueAt( int t );
  void SetValueAt( int t, float v );
};
```

We will assume that the `GetValueAt()` operation will block until the value of the cell for the requested time is available. A cellular automaton is constructed by creating an array and initialising it with cells. The cell values are initially zero, with two high-value "peaks" being placed manually towards the centre of the model. This may be accomplished as follows:

```
Array *a;   Cell *cell;
Point p(2, 0, 0), q(2, 10000, 10000);
Region *r =new Region(p, q);

cell=new Cell;   cell->SetValueAt(0, 0.0);
a=new Array(r);   a->Initialise(cell);

cell->SetValueAt(0, 5000.0);
p.Is(100, 5000);   a->AtCopy(p, cell);
p.Is(9900, 5000);   a->AtCopy(p, cell);
```

In this code fragment, a `Cell` is first created and initialised to have a value of zero at time zero. An `Array` is then created using the `new` operator, and it partitoned by the call to `Initialise()`. This operation will both diistribute the array's storage and initialise each element with a copy of the given `Cell`. The two peaks are inserted by placing copies of the appropriate-valued `Cell` into the array where desired – the first probably local, the second almost certainly remote.

Internally, the access routines `AtPut()`, `AtCopy()` and `At()` perform resolution to acquire the holder of the possibly remote elements. The resolution takes the form of a locality test possibly followed by a call to the partition object and a re-direction of the request, with "hidden" functions being called to manipulate local storage (the `BasicAt()` and `BasicAtPut()` routines in the class description):

```
void ObjectArray::AtPut( Point p, Object *o ) {
 ObjectArray *a;

 a=(ObjectArray *) BasicResolve(p);
 if(a==this) {
  /* store element locally */
  BasicAtPut(p, &o);
 } else {
  /* forward request to appropriate component */
  c->AtPut(p, o);
 }
}
```

Different sub-classes may use storage representations without affecting the user-level routine, simply by over-riding the basic storage manipulation functions.

Concurrency

Sequential access, whilst managing distribution, does not fully utilise the potential parallelism of partitioning. If parallel processing is required, a process object must be instantiated and attached to the collection. The `Body()` routine of such a task is shown below:

```
void CellActivity::Body( void ) {
 Array *a =(Array *) GetAttachment();
 Region *r =a->GetLocalRegion();
 Point p(2), q(2), dp(2);
 Iterator iter =r->NewIterator();
 int x, y;   int t;
 float value;
 Cell *cell, *n;

 for(t=0; t<simTime; t++) {
  p=r->First(iter);
  while(!p.Undefined()) {
   cell=a->At(p);   value=0.0;
   for(x=-1; x<=1; i++)
    for(y=-1; y<=1; y++) {
     dp.Is(x, y);   q=p;   q.Add(dp);
     n=a->At(q);
     if(n!=nil) value+=n->GetValueAt(t);
    }

   cell->SetValueAt(t+1, value/9);
   p=r->Next(iter);
  }
 }
}
```

At each time step, a cell's value for the next step is set to the average of its neighbour's values, with an activity dealing with the local elements of the component to which it is attached. Instances of the activity may be attached to a structure and started in parallel:

```
a->AttachStartAndAwait(new CellActivity);
```

In this example, the caller will block until all the workers have finished: by using the `AttachAndStart()` routine instead of `AttachStartAndAwait()`, the activities could left to execute independently. Both these routines are simplifications of the `Attach()` routine, which returns the `Group` of activities being used by the collection. The process bodies will then execute until the simulation time ends, whereupon the array may be queried to determine the values of the cells.

It should be noted that there is no explicit statement anywhere as to the distribution of the `Array` or as to the amount of concurrent activity deployed to process it. It is the system's responsibility to ensure that these factors are dealt with.

Extensibility

A class library is only useful if it easily extensible, so programmers can create new structures from the supplied infrastructure. Furthermore, it is vital that programmers writing dedicated Phoenix collections are freed from low-level distribution concerns – this is as important for sub-classes as for external clients.

The important point is that, in many respects, a sub-class behaves *exactly* as a client, and may hence use the parent's public interface. This means that distribution is transparent to sub-classes of collection classes, as much as to clients. The only time when a sub-class needs to be concerned with distribution is when it uses “covert” knowledge to manipulate its parent's internal data structures in the interests of performance.

As an example, we will derive a "well-behaved" sub-class of `ObjectArray` tailored towards our cellular automaton. It defines an additional function which calculates the average values of cells around a particular cell, freeing client objects of the need to perform averaging from outside:

```
class CellArray : public ObjectArray {
public:
 CellArray( Region *gr );

 float GetAverageOf( Point p );
};

float CellArray::GetAverageOf( Point p ) {
 int x, y;   float value;
 Point dp(2), q(2);   Cell *cell;

 value=0.0;
 for(x=-1; x<=1; i++)
  for(y=-1; y<=1; y++) {
   dp.Is(x, y);   q=p;   q.Add(dp);
   cell=At(q);
   if(cell!=nil) value+=cell->GetValueAt(t);
  }

 return value/9;
}
```

This new sub-class makes use of the standard `At()` routine, which handles all resolution. The `GetAverage()` routine simply uses this routine to access the data it requires, without any knowledge of distribution.

It is similarly quite simple to use a novel partitioning strategy. For arrays, one may use a sub-class of `Region` which, for example, defines a hexagonal region (useful in some other kinds of cellular automata). By defining appropriate virtual members for the new region (as illustrated in figure 2), it may be used to distribute an `Array` without any intervention on the part of the collection class – although the data held locally by each component will change.

In a similar way one might also generate specialised partition sub-classes for particular applications. An example might be an `Array` partition which maps components onto an *n*x*m* mesh, one per processor, exactly as might occur using a traditional data mapping tool.

Constraints of space prevent us from discussing these points more fully: however the partitioned model, by breaking-down the responsibilities of classes between data manipulation and distribution, simplifies the task of creating distributed applications without preventing the knowledgeable programmer from fully controlling an application's mapping onto the hardware.

6. Conclusion

From the outset we have been considering the use of parallel computers as *general-purpose* computing engines, rather than as supercomputers. We believe that scalable parallel machines built from "off the shelf" components promise to make supercomputer levels of performance available to a wider class of users than is currently possible – but for this to occur, the programming interfaces to such machines must be simplified.

We have described a programming environment whose main goal is to allow applications to exploit the scalability of the underlying parallel machine. It approaches this by hiding the disjoint-memory topology of the machine behind behind a set of scalable distributed data structures. These structures may be used both to arrange objects in memory and to regulate concurrency, without the programmer being forced to consider either the exact distribution used or the amount of concurrency to deploy. Run-time properties of objects allow the details of distribution – the size of components and the like – to be tailored to improve performance without re-compilation.

By separating the concerns of data *versus* distribution management through the use of parallel class hierachies, our system allows one aspect of an application to be changed without affecting the other. This means that programs may be written and debugged before their exact distribution is decided upon.

We are currently proceeding with the evaluation of Phoenix to determine both its efficiency and appropriateness to general-purpose scalable parallel programming. Proposed future work involves the investigation of different partitioning strategies and their dynamics, especially in the case where several structures interact by sharing a set of processors. We are also interested in the effects of placing processes themselves into partitioned collections, using the collections as a process as well as data structuring technique.

We believe that scalable systems offer major advantages for available, high-performance computing in the future, and that the key to their successful exploitation lies in providing high-level views of their capabilities. Partitioning is an attempt to provide these features.

7. References

[1] Sudhir Ahuja, N. Carriero and D. Gelernter, "*Linda and friends*", IEEE Computer **19**(8) (August 1986), pp. 26-34.

[2] Colin Atkinson, *Object-oriented reuse, concurrency and distribution*, Addison-Wesley, 1991.

[3] Paul B. Austin, Kevin A. Murray and Andy J. Wellings, "*The design of an operating system for a scalable parallel computing engine*", SOFTWARE – practice and experience **21**(10) (October 1991), pp. 989-1013.

[4] Paul B. Austin, Kevin A. Murray and Andy J. Wellings, "*File system caching in large point-to-point networks*", Software engineering journal **7**(1) (January 1992), pp. 65-80.

[5] A.D. Birrell and B.J. Nelson, "*Implementing remote procedure call*", ACM Transactions on computer systems **2**(1) (February 1984), pp. 39-59.

[6] David Bruce, "*A strongly-typed approach to parallel systems*", pp. 57-59 in Collected position papers of the BCS workshop on abstract machine models for highly parallel computers, volume 2, University of Leeds, 25-27th March, 1991.

[7] Andrew A. Chien and William J. Dally, "*Concurrent Aggregates*", ACM SIGPLAN Notices **25**(3) (March 1990), pp. 187-196. Proceedings of the 2nd ACM symposium on principles and practice of parallel programming.

[8] Simon A. Dobson and Andy J. Wellings, "*Programming highly parallel general-purpose applications (extended abstract)*", pp. 49-56 in Collected position papers of the BCS workshop on abstract machine models for highly parallel computers, volume 2, University of Leeds, 25-27th March, 1991.

[9] R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, "*Extendible hashing: a fast access method for dynamic files*", ACM Transactions of database systems **4**(3) (September 1979), pp. 315-344.

[10] E. Fredkin, "*Trie memory*", Communications of the ACM **3**(9) (September 1960), pp. 439-499.

[11] Adele Goldberg and David Robson, "Smalltalk-80: the language and its implementation", Addison-Wesley, 1985.

[12] R.H. Halstead, "*MultiLisp: a language for concurrent symbolic computation*", ACM Transactions on programming languages and systems **7**(4) (October 1985), pp. 501-538.

[13] E. Jul, H. Levy, N. Hutchinson and A. Black, "*Fine-grained mobility in the Emerald system*", ACM Transactions on computer systems **6**(1) (February 1988), pp. 109-133.

[14] Per-Åke Larson, "*Dynamic hashing*", BIT **18** (1978), pp. 184-201.

[15] Kai Li, "*Shared virtual memory on loosely coupled multiprocessors*", Department of computer science, Yale University, September 1986. Ph.D. dissertation.

[16] B. Liskov and L. Shira, "*Promises: linguistic support for efficient asynchronous procedure calls*", ACM SIGPLAN Notices **23**(7) (July 1988). Proceedings of the ACM conference on programming language design and implementation.

[17] Witold Litwin, "*Virtual hashing: a dynamically changing hashing*", pp. 517-523 in Proceedings of the 4th international conference on very large data bases, 1978.

[18] S.E. Lucco, "*Parallel programming in a virtual object space*", ACM SIGPLAN Notices **22**(12) (December 1987), pp. 26-34. Proceedings of OOPSLA '87.

[19] David May, "*Occam-2 language definition*", Inmos, 13 February 1987.

[20] Kevin A. Murray and Andy J. Wellings, "*Issues in the design of a distributed operating system for a network of Transputers*", Microprocessing and microprogramming **24**(1-5) (1988), pp. 169-178.

[21] Kevin A. Murray and Andy J. Wellings, "*Wisdom: a distributed operating system for Transputers*", Computer systems science and engineering **5**(1) (January 1990), pp. 13-20.

[22] Kevin A. Murray, "*Wisdom: the foundation of a scalable parallel operating system*", Department of Computer Science, University of York, February 1990. Ph.D. dissertation.

[23] Graham D. Parrington, "*Management of concurrency in a reliable object-oriented computing system*", Computing laboratory, University of Newcastle upon Tyne, July 1988. Ph.D. dissertation.

[24] Bjarne Stroustrup, "*The C++ programming language*", Addison-Wesley, 1987.

[25] Greg Wilson, "*The life and times of cellular automata*", New Scientist (8th October 1988), pp. 44-47.